OpenMM Preview Release 4

Peter Eastman
August 19, 2009

Goals and Design Principles

OpenMM is an API for executing molecular dynamics simulations on high performance computer architectures. Examples of the sorts of architectures it is intended to support include:

- Highly parallel systems with large numbers of CPU cores
- Graphics processing units (GPUs)
- Clusters of computers communicating over a network

The target audience for the API is developers of simulation software. It is explicitly *not* targeted at computational biologists or other people who want to run simulations. They will continue to use the same software packages they currently do. OpenMM is targeted at the developers of those packages, and offers them a way to easily take advantage of a variety of high performance architectures.

The design of the OpenMM API is guided by the following principles.

1. The API should be narrow in scope.

We have intentionally restricted the API to only those features which directly support the goal stated above: allowing developers of simulation software to support high performance architectures. For example, it does not include any routines for reading or writing files, any model building utilities, or any analysis features. These are outside the scope of OpenMM. Any simulation package will certainly need the ability to read files, build molecular models based on them, etc., but OpenMM does not help with those tasks. It does not try to solve all possible problems, only to solve one particular problem well.

2. The API must support efficient implementations on a variety of architectures.

The most important consequence of this goal is that the API cannot provide direct access to state information (particle positions, velocities, etc.) at all times. On some architectures, accessing this information is expensive. With a GPU, for example, it will be stored in video memory, and must be transferred to main memory before outside code can access it. On a distributed architecture, it might not even be present on the local computer. OpenMM therefore only allows state information to be accessed in bulk, with the understanding that doing so may be a slow operation.

3. The API should be easy to understand and easy to use.

This seems obvious, but it is worth stating as an explicit goal. We are creating OpenMM with the hope that many other people will use it. To achieve that goal, it should be possible for someone to learn it without an enormous amount of effort. An equally important aspect of being "easy to use" is being easy to use *correctly*. A well designed API should minimize the opportunities for a programmer to make mistakes. For both of these reasons, clarity and simplicity are essential.

4. It should be modular and extensible.

We cannot hope to provide every feature any user will ever want. For that reason, it is important that OpenMM be easy to extend. If a user wants to add a new molecular force field, a new thermostat algorithm, or a new hardware platform, the API should make that easy to do.

5. The API should be hardware independent.

Computer architectures are changing rapidly, and it is impossible to predict what hardware platforms might be important to support in the future. One of the goals of OpenMM is to separate the API from the hardware. The developers of a simulation application should be able to write their code once, and have it automatically take advantage of any architecture that OpenMM supports, even architectures that do not yet exist when they write it.

Choice of Language

Molecular modeling and simulation tools are written in a variety of languages: C, C++, Fortran, Python, TCL, etc. It is important that any of these tools be able to use OpenMM. There are two possible approaches to achieving this goal.

One option is to provide a separate version of the API for each language. These could be created by hand, or generated automatically with a wrapper generator such as SWIG. This would require the API to use only "lowest common denominator" features that can be reasonably supported in all languages. For example, an object oriented API would not be an option, since it could not be cleanly expressed in C or Fortran.

The other option is to provide a single version of the API written in a single language. This would permit a cleaner, simpler API, but also restrict the languages it could be directly called from. For example, a C++ API could not be invoked directly from Fortran or Python.

We have chosen to use a hybrid of these two approaches. OpenMM is based on an object oriented C++ API. This is the primary way to invoke OpenMM, and is the only API that fully exposes all features of the library. We believe this will ultimately produce the best,

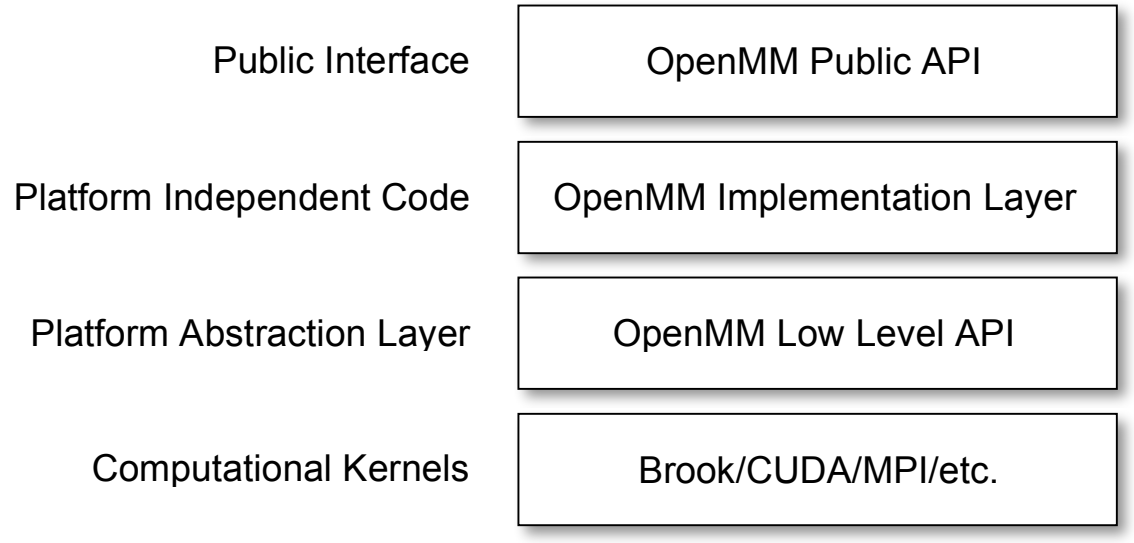
easiest to use API and create the least work for developers who use it. It does require that any code which directly invokes this API must itself be written in C++, but this should not be a significant burden. Regardless of what language we had chosen, developers would need to write a thin layer for translating between their own application's data model and OpenMM. That layer is the only part which needs to be written in C++.

In addition, we have created wrapper APIs that allow OpenMM to be invoked from other languages. The current release includes wrappers for C and Fortran, and a Python wrapper is under development. These wrappers support as many features as reasonably possible given the constraints of the particular languages, but some features cannot be fully supported. In particular, writing plugins to extend the OpenMM API can only be done in C++.

We are also aware that some features of C++ can easily lead to compatibility and portability problems, and we have tried to avoid those features. In particular, we make minimal use of templates, avoid multiple inheritance altogether, and use exceptions only to report user errors, never within computational kernels. Our goal is to eventually support OpenMM on all major compilers and operating systems.

Architectural Overview

OpenMM is based on a layered architecture, as shown in the following diagram:



At the highest level is the OpenMM public API. This is the API developers program against when using OpenMM within their own applications. It is designed to be simple, easy to understand, and completely platform independent. This is the only layer that many users will ever need to look at.

The public API is implemented by a layer of platform independent code. It serves as the interface to the lower level, platform specific code. Most users will never need to look at it.

The next level down is the OpenMM Low Level API (OLLA). This acts as an abstraction layer to hide the details of each hardware platform. It consists of a set of C++ interfaces that each platform must implement. Users who want to extend OpenMM will need to write classes at the OLLA level. Note the different roles played by the public API and the low level API: the public API defines an interface for users to invoke in their own code, while OLLA defines an interface that users must implement, and that is invoked by the OpenMM implementation layer.

At the lowest level is hardware specific code that actually performs computations. This code may be written in any language and use any technologies that are appropriate. For example, code for GPUs will be written in stream processing languages such as Brook or CUDA, code written to run on clusters will use MPI or other distributed computing tools, code written for multicore processors will use threading tools such as Pthreads or OpenMP, etc. OpenMM sets no restrictions on how these computational kernels are written. As long as they are wrapped in the appropriate OLLA interfaces, OpenMM can use them.

The OpenMM Public API

The public API is based on a small number of classes:

System: A System specifies the number of particles to be simulated, the mass of each one, and any constraints between them. The interactions between the particles are specified through a set of Force objects (see below) that are added to the System. Force field specific parameters, such as particle charges, are not direct properties of the System. They are properties of the Force objects contained within the System.

Force: The Force objects added to a System define the behavior of the particles. Force is an abstract class; subclasses implement specific behaviors. The Force class is actually slightly more general than its name suggests. A Force can, indeed, apply forces to particles, but it can also directly modify particle positions and velocities in arbitrary ways. Some thermostats and barostats, for example, can be implemented as Force classes. (We recognize that Force is a somewhat misleading name for the class. Suggestions of better names are welcome!)

OpenMM Preview Release 3 provides eight Force subclasses: HarmonicBondForce, HarmonicAngleForce, PeriodicTorsionForce, RBTorsionForce, NonbondedForce, GBSAOBCForce, AndersenThermostat, and CMMotionRemover.

Context: This stores all of the state information for a simulation: particle positions and velocities, as well as arbitrary parameters defined by the Forces in the System. It is

possible to create multiple Contexts for a single System, and thus have multiple simulations of that System in progress at the same time.

Integrator: This implements an algorithm for advancing the simulation through time. It is an abstract class; subclasses implement specific algorithms. OpenMM Preview Release 3 provides five Integrator subclasses: LangevinIntegrator, VerletIntegrator, BrownianIntegrator, VariableLangevinIntegrator, and VariableVerletIntegrator.

State: A State stores a snapshot of the simulation at a particular point in time. It is created by calling a method on a Context. As discussed earlier, this is a potentially expensive operation. This is the only way to query the values of state variables, such as particle positions and velocities; Context does not provide methods for accessing them directly.

Here is an example of what the source code to create a System and run a simulation might look like:

```
System system;
for (int i = 0; i < numParticles; ++i)
    system.addParticle(particle[i].mass);
HarmonicBondForce* bonds = new HarmonicBondForce();
system.addForce(bonds);
for (int i = 0; i < numBonds; ++i)
    bonds->addBond(bond[i].particle1, bond[i].particle2,
        bond[i].length, bond[i].k);
HarmonicAngleForce* angles = new HarmonicAngleForce();
system.addForce(angles);
for (int i = 0; i < numAngles; ++i)
    angles->addAngle(angle[i].particle1, angle[i].particle2,
        angle[i].particle3, angle[i].angle, angle[i].k);
// ...create and initialize other force field terms in the same way
LangevinIntegrator integrator(temperature, friction, stepSize);
Context context(system, integrator);
context.setPositions(initialPositions);
context.setVelocities(initialVelocities);
integrator.step(10000);
```

We create a System, add various Forces to it, and set parameters on both the System and the Forces. We then create a LangevinIntegrator, initialize a Context in which to run a simulation, and instruct the Integrator to advance the simulation for 10,000 time steps.

The OpenMM Low Level API

The OpenMM Low Level API (OLLA) defines a set of interfaces that users must implement in their own code if they want to extend OpenMM, such as to create a new Force subclass or support a new hardware platform. It is based on a stream processing architecture: “kernels” define computations that are performed on “streams” of data, and possibly write their results to other streams.

More specifically, there are two abstract classes called **KernelImpl** and **StreamImpl**. Instances of these classes (or rather, of their subclasses) are created by **KernelFactory** and **StreamFactory** objects. These classes provide the concrete implementations of streams and kernels for a particular platform. For example, to perform calculations on a GPU, one would create one or more StreamImpl subclasses that stored the stream data in video memory on the GPU, one or more KernelImpl subclasses that implemented the computations with GPU kernels, and one or more KernelFactory and StreamFactory subclasses to instantiate the KernelImpl and StreamImpl objects.

All of these objects are encapsulated in a single object that extends **Platform**. StreamFactory and KernelFactory objects are registered with the Platform to be used for creating specific named kernels and streams. The choice of what implementation to use (a GPU implementation, a multithreaded CPU implementation, an MPI-based distributed implementation, etc.) consists entirely of choosing what Platform to use.

As discussed so far, the low level API is not in any way specific to molecular simulation; it is a fairly generic stream processing API. In addition to defining the generic classes, OpenMM also defines abstract subclasses of KernelImpl corresponding to specific calculations. For example, there is a class called CalcHarmonicBondForceKernel to implement HarmonicBondForce and a class called IntegrateLangevinStepKernel to implement LangevinIntegrator. It is these classes for which each Platform must provide a concrete subclass.

This architecture is designed to allow easy extensibility. To support a new hardware platform, for example, you create concrete subclasses of all the abstract kernel classes, then create appropriate factories and a Platform subclass to bind everything together. Any program that uses OpenMM can then use your implementation simply by specifying your Platform subclass as the platform to use.

Alternatively, you might want to create a new Force subclass to implement a new type of interaction. To do this, define an abstract KernelImpl subclass corresponding to the new force, then write the Force class to use it. Any Platform can support the new Force by providing a concrete implementation of your KernelImpl subclass. Furthermore, you can easily provide that implementation yourself, even for existing Platforms created by other people. Simply create a new KernelFactory subclass for your kernel and register it with the Platform object. The goal is to have a completely modular system. Each module, which might be distributed as an independent library, can either add new features to existing platforms or support existing features on new platforms.

In fact, there is nothing "special" about the kernel classes defined by OpenMM. They are simply KernelImpl subclasses that happen to be used by Forces and Integrators that happen to be bundled with OpenMM. They are treated exactly like any other KernelImpl, including the ones you define yourself.

It is important to understand that OLLA defines an interface, not an implementation. It would be easy to assume a one-to-one correspondence between KernelImpl objects and the pieces of code that actually perform calculations, but that need not be the case. For a GPU implementation, for example, a single KernelImpl might invoke several GPU kernels. Alternatively, a single GPU kernel might perform the calculations of several KernelImpl subclasses. Similarly, StreamImpl defines only an API for storing and retrieving data; it sets no restrictions on how that data should be stored.

Platforms

OpenMM Preview Release 4 contains the following Platform subclasses:

- ReferencePlatform. This is designed to serve as reference code for writing other platforms. It is written with simplicity and clarity in mind, not performance.
- CudaPlatform. This platform is implemented using the CUDA language, and performs calculations on Nvidia GPUs.
- BrookPlatform. This platform is implemented using the Brook language, and performs calculations on ATI GPUs. It is available only on Windows. This platform is deprecated, and many features are not supported on it. We eventually plan to replace it with an OpenCL based platform.

Future Plans

There are a few features not present in Preview Release 4 that we hope to include in version 1.0. The most important of these are particle-mesh Ewald and support for forces with arbitrary, user defined functional forms.

We also expect a variety of API changes to occur as development continues. These changes are unlikely to require dramatic changes to invoking code, but users should expect that programs written to use this release will require modifications to work with version 1.0.

Finally, we are open to other possible changes. All comments and suggestions are welcome for ways to make OpenMM a better, more useful toolkit. Email us at openmm-team@simtk.org.

Referencing OpenMM

Any work that uses OpenMM should cite the following publication:

M. S. Friedrichs, P. Eastman, V. Vaidyanathan, M. Houston, S. LeGrand, A. L. Beberg, D. L. Ensign, C. M. Bruns, V. S. Pande. "Accelerating Molecular Dynamic Simulation on Graphics Processing Units." J. Comp. Chem., 30(6):864-872 (2009).

Acknowledgements and License

OpenMM was developed by Simbios, the NIH National Center for Physics-Based Simulation of Biological Structures at Stanford, funded under the NIH Roadmap for Medical Research, grant U54 GM072970. See https://simtk.org.

Portions copyright © 2008-2009 Stanford University and the Authors.

Two different licenses are used for different parts of OpenMM. The public API, the low level API, and the reference platform are all distributed under the MIT license. This is a very permissive license which allows them to be used in almost any way, requiring only that you retain the copyright notice and disclaimer when distributing them.

The CUDA and Brook platforms are distributed under the GNU Lesser General Public License (LGPL). This also allows you to use, modify, and distribute them in any way you want, but it requires you to also distribute the source code for your modifications. This restriction applies only to modifications to OpenMM itself; you need not distribute the source code to applications that use it.

OpenMM also uses several pieces of code that were written by other people and are covered by other licenses. All of these licenses are similar in their terms to the MIT license, and do not significantly restrict how OpenMM can be used.

All of these licenses may be found in the "licenses" directory included with OpenMM.